Reflective PE Loading

By Gregory Basior

BLUF: A reflectively loaded Portable Executable (PE) is a windows exe or dll that can be executed without creating a new process or loading a dll. The purpose of loading a PE in this manner is to write the file directly to memory and execute it in order to gain the privileges of the existing process and avoid antivirus detection. The following will describe the technical details related to how a 32-bit PE can be loaded reflectively.

Note - All of the disassembly displayed is from a basic stageless payload created by msfvenom.

Step 1: Position Independent Code

The PE file will be injected into the memory space of a running process. It will be injected in such a way that it is written directly from how it is stored in memory into a buffer. The file can not be run in this manner because all function imports must be resolved and all data that is part of a relocation table must be resolved. In order to do all of this, the file must know where it is located in memory. In order to do this, there will be a calculation placed at the beginning of the PE file:

```
00416000 4d             DEC     EBP
00416001 5a             POP     EDX
00416002 e8 00 00       CALL    LAB_00416007
         00 00

                  LAB_00416007
00416007 5b             POP     EBX
00416008 52             PUSH    EDX
00416009 45             INC     EBP
0041600a 55             PUSH    EBP
0041600b 89 e5          MOV     EBP,ESP
0041600d 81 c3 64       ADD     EBX,0x1364
         13 00 00
00416013 ff d3          CALL    EBX=>FUN_0041736b
```

Notice that this is the beginning of a PE file as it begins with `MZ`. Those two bytes will be run as a meaningless command, but after that it does the following:

`CALL 0x00000000` - This will push the next address onto the stack and also jump to that address.

`POP EBX` - This will store the location of the current address into EBX.

`ADD EBX,0x1364`
`CALL EBX` - These commands will now go to the location of the reflective loader.

Step 2: Find module base

The reflective loader entry point will search for the module base although it could be determined in the previous step. In order to do so, it will first find the current address, which can be done in a similar manner through a CALL / POP set of commands.

```
                     FIND_MODULE_BASE
0041738c b8 4d 5a        MOV        EAX,0x5a4d
         00 00
00417391 66 39 06        CMP        word ptr [ESI],AX
00417394 75 17           JNZ        LAB_004173ad
00417396 8b 46 3c        MOV        EAX,dword ptr [ESI + 0x3c]
00417399 8d 48 c0        LEA        ECX,[EAX + -0x40]
0041739c 81 f9 bf        CMP        ECX,0x3bf
         03 00 00
004173a2 77 09           JA         LAB_004173ad
004173a4 81 3c 30        CMP        dword ptr [EAX + ESI*0x1],0x4550
         50 45 00 00
004173ab 74 03           JZ         FOUND_BASE

                     LAB_004173ad
004173ad 4e              DEC        ESI
004173ae eb dc           JMP        FIND_MODULE_BASE
```

In order to find the module base, there will be a loop beginning at the current address that will decrement that address until it finds the base.
First it will check to see if the word at the current address is `0x5a4d`, which is the MZ header.

`MOV EAX, DWORD PTR [ESI+0X3C]` - If this is the MZ header, this will be the value of `e_lfanew`, which would hold the PE header.

`LEA ECX, [EAX - 0X40]`
`CMP ECX, 0X3BF` - Check to see if the value is between 0x40 and 0x400. The value cannot be less than 0x40 because that is the size of the DOS header.

`CMP DWORD PTR [EAX + ESI * 0X1], 0X4550` - Check to ensure that the value at the offset is actually the PE header.

If all of these conditions are met, we now have the base address of the module.

Step 3: Find necessary functions

Find necessary modules

In order to completely load this PE file, its import table will need to be resolved. In order to do this, all external DLLs must be loaded and the address of the imported functions must be found. The reflective loader needs to find the address of some functions that are already loaded in memory by the parent process. These functions are located in `kernel32.dll` and `ntdll.dll`. The functions are:

- `NtFlushInstructionCache`
- `GetProcAdress`
- `LoadLibraryA`
- `VirtualAlloc`
- `VirtualLock`

```
004173b0 64 a1 30       MOV     EAX,FS:[0x30]
         00 00 00
004173b6 89 75 fc       MOV     dword ptr [EBP + local_8],ESI
004173b9 c7 45 d0       MOV     dword ptr [EBP + local_34],0x2
         02 00 00 00
004173c0 c7 45 d4       MOV     dword ptr [EBP + local_30],0x1
         01 00 00 00
004173c7 8b 40 0c       MOV     EAX,dword ptr [EAX + 0xc]
004173ca 8b 58 14       MOV     EBX,dword ptr [EAX + 0x14]
004173cd 89 5d f0       MOV     dword ptr [EBP + local_14],EBX
```

`MOV EAX, FS:[0X30]` - Get the address of Process Environment Block (PEB).

`MOV EAX, DWORD PTR [EAX + 0XC]` – Get the address of the PPEB_LDR_DATA.

`MOV EBX, DWORD PTR [EAX + 0X14]` – Get the address of the InMemoryModuleList. This is a linked list of LDR_DATA_TABLE_ENTRY elements. The elements point to the next and previous elements of the list. They also contain a UNICODE_STRING structure that contains the name of the module and a pointer to the base address of the module.

```
004173d8 8b 53 28        MOV        EDX,dword ptr [EBX + 0x28]
004173db 33 c9           XOR        ECX,ECX
004173dd 0f b7 7b 24     MOVZX      EDI,word ptr [EBX + 0x24]

                     LAB_004173e1
004173e1 8a 02           MOV        AL,byte ptr [EDX]
004173e3 c1 c9 0d        ROR        ECX,0xd
004173e6 3c 61           CMP        AL,0x61
004173e8 0f b6 c0        MOVZX      EAX,AL
004173eb 72 03           JC         LAB_004173f0
004173ed 83 c1 e0        ADD        ECX,-0x20

                     LAB_004173f0
004173f0 03 c8           ADD        ECX,EAX
004173f2 81 c7 ff        ADD        EDI,0xffff
         ff 00 00
004173f8 42              INC        EDX
004173f9 66 85 ff        TEST       DI,DI
004173fc 75 e3           JNZ        LAB_004173e1
004173fe 81 f9 5b        CMP        ECX,0x6a4abc5b
```

It will then loop through the elements of this list until it finds the module of interest.

`MOV EDX, DWORD PTR [EBX + 0X28]` - This is the offset to the buffer containing the unicode string describing the module, e.g. kernel32.dll.

`XOR ECX, ECX` - This will store a hash of the unicode string.

`MOV EDI, WORD PTR [EBX + 0X24]` - This is the size of the unicode string.

The loop at location 0x004173e1 does the following:

- Get the next byte in the unicode string.
- Convert the byte to upper case.
- Rotate right the current hash by 0xD bytes.
- Add the current byte to the current hash.
- Continue until the number of bytes read is equal to the size of the string.

`CMP ECX, 0X6A4ABC5B` - This is checking to see if the hash is equal to the hash of KERNEL32.DLL. This process will continue in a loop through the modules until this module is found.

Find function within module

```
0041740a 8b 7b 10      MOV     EDI,dword ptr [EBX + 0x10]
0041740d c7 45 f8      MOV     dword ptr [EBP + local_c],0x4
         04 00 00 00
00417414 8b 47 3c      MOV     EAX,dword ptr [EDI + 0x3c]
00417417 8b 44 38 78   MOV     EAX,dword ptr [EAX + EDI*0x1 + 0x78]
0041741b 03 c7         ADD     EAX,EDI
0041741d 89 45 d8      MOV     dword ptr [EBP + local_2c],EAX
00417420 8b 70 20      MOV     ESI,dword ptr [EAX + 0x20]
00417423 8b 40 24      MOV     EAX,dword ptr [EAX + 0x24]
00417426 03 f7         ADD     ESI,EDI
00417428 03 c7         ADD     EAX,EDI
0041742a 89 45 f4      MOV     dword ptr [EBP + local_10],EAX
0041742d 8b d8         MOV     EBX,EAX
```

`MOV EDI, DWORD PTR [EBX+0X10]` - EBX still contains the value of the current LDR_DATA_TABLE_ENTRY. This command loads the base address of the module found.

`MOV EAX, DWORD PTR [EDI+0X3C]` - Get the offset of the PE header.

`MOV EAX, DWORD PTR [EAX + EDI*0X1 + 0X78]`
`ADD EAX, EDI` - Get the relative virtual address (RVA) of the export table and add that to the base address to get the in memory address of the export table.

`MOV ESI, DWORD PTR [EAX+0X20]` - Get the RVA of the name pointer table.

`MOV EAX, DWORD PTR [EAX+0X24]` - Get the RVA of the ordinal table.

`ADD ESI, EDI`
`ADD EAX, EDI` - Get the in memory address of the name pointer and ordinal table. The name pointer table is an array of RVA's that point to the ascii string names of the functions. The ordinal table is an array of words that can translate from an offset in the name table to an offset in the address table.

```
0041742f 8b 0e        MOV     ECX,dword ptr [ESI]
00417431 03 cf        ADD     ECX,EDI
00417433 33 d2        XOR     EDX,EDX
00417435 8a 01        MOV     AL,byte ptr [ECX]

                 LAB_00417437
00417437 c1 ca 0d     ROR     EDX,0xd
0041743a 0f be c0     MOVSX   EAX,AL
0041743d 03 d0        ADD     EDX,EAX
0041743f 41           INC     ECX
00417440 8a 01        MOV     AL,byte ptr [ECX]
00417442 84 c0        TEST    AL,AL
00417444 75 f1        JNZ     LAB_00417437
```

Once it has this information, it will calculate the hash of the current name. The only difference between this and calculating the hash of the module is that it does not change the case.

```
004174c3 8b 45 f8     MOV     EAX,dword ptr [EBP + COUNTER]

                 LAB_004174c6
004174c6 6a 02        PUSH    0x2
004174c8 59           POP     ECX
004174c9 83 c6 04     ADD     ESI,0x4
004174cc 03 d9        ADD     EBX,ECX
004174ce 66 85 c0     TEST    AX,AX
004174d1 0f 85 58     JNZ     LAB_0041742f
```

After it completes the current name, it will continue looping through the table until there are no names left or it has found the predetermined number of names.

`MOV EAX, DOWRD PTR [EBP+COUNTER]` - In the case of kernel32.dll, this value is initialized to 4 and is decremented everytime a relevant function name is found. The loop will exit when this reaches 0.

`ADD ESI, 0X4` - This will increment to the next RVA in the name array.

`ADD EBX, ECX` - This will increment the array of words containing ordinals by 2.

```
00417466 8b 45 d8        MOV      EAX,dword ptr [EBP + EXPORT_TABLE]
00417469 0f b7 0b        MOVZX    ECX,word ptr [EBX]
0041746c 8b 40 1c        MOV      EAX,dword ptr [EAX + 0x1c]
0041746f 8d 04 88        LEA      EAX,[EAX + ECX*0x4]
00417472 03 c7           ADD      EAX,EDI
```

When it finds a matching hash, the above code block will be hit.

`MOV EAX, DWORD PTR [EBP + EXPORT_TABLE]` – Get the address of the export table.

`MOVZX ECX, WORD PTR [EBX]` - Get the current ordinal value.

`MOV EAX, DWORD PTR [EAX + 0X1C]` - Get the RVA of the export address table.

`LEA EAX, [EAX + ECX*0X4]` – Get the value of the offset in the export address table. This is the value of the ordinal multiplied by 4 plus the RVA of the export address table.

`MOV EAX, EDI` - Get the in address value of the location of the RVA of the current function.

```
0041748d 8b 00           MOV      EAX,dword ptr [EAX]
0041748f 03 c7           ADD      EAX,EDI
00417491 89 45 e0        MOV      dword ptr [EBP + local_24],EAX
```

Finally, store the address of the function.

`MOV EAX, DWORD PTR [EAX]` - Get the RVA located at the address inside of the export table.

`ADD EAX, EDI` - Add the RVA to the base address to get the in memory address of that function.

`MOV DWORD PTR [EBP + LOCAL_24], EAX` - Store the function address in a local variable to be called later.

Step 4: Allocate memory to write the loaded image

```
00417585 8b 75 fc        MOV        ESI,dword ptr [EBP + local_8]

                     LAB_00417588
00417588 8b 5e 3c        MOV        EBX,dword ptr [ESI + 0x3c]
0041758b 6a 40           PUSH       0x40
0041758d 03 de           ADD        EBX,ESI
0041758f 68 00 30        PUSH       0x3000
         00 00
00417594 89 5d f0        MOV        dword ptr [EBP + PE_HEADER],EBX
00417597 ff 73 50        PUSH       dword ptr [EBX + 0x50]
0041759a 6a 00           PUSH       0x0
0041759c ff 55 ec        CALL       dword ptr [EBP + VirtualAlloc]
0041759f ff 73 50        PUSH       dword ptr [EBX + 0x50]
004175a2 8b f8           MOV        EDI,EAX
004175a4 57              PUSH       EDI
004175a5 89 7d f4        MOV        dword ptr [EBP + NEW_IMAGE],EDI
004175a8 ff 55 e8        CALL       dword ptr [EBP + VirtualLock]
```

`MOV EBX, DWORD PTR [ESI + 0x3C]`
`ADD EBX, ESI` - Get the address of the PE header.

`PUSH 0X40` - Make the new memory PAGE_EXECUTE_READWRITE.
`PUSH 0X3000` - Commit the new memory.
`PUSH DWORD PTR [EBX + 0X50]` - Make the new memory the SizeOfImage, taken from the PE header.

Call VirtualAlloc followed by VirtualLock to ensure there are no page faults when accessing the memory.

Step 5: Move headers into new image

```
004175ab 8b 53 54        MOV        EDX,dword ptr [EBX + 0x54]
004175ae 8b ce           MOV        ECX,ESI
004175b0 85 d2           TEST       EDX,EDX
004175b2 74 12           JZ         LAB_004175c6
004175b4 8b c7           MOV        EAX,EDI
004175b6 2b c6           SUB        EAX,ESI
004175b8 89 45 d8        MOV        dword ptr [EBP + TEMP_VAR],EAX
004175bb 8b f0           MOV        ESI,EAX

                     MEM_CPY
004175bd 8a 01           MOV        AL,byte ptr [ECX]
004175bf 88 04 0e        MOV        byte ptr [ESI + ECX*0x1],AL
004175c2 41              INC        ECX
004175c3 4a              DEC        EDX
004175c4 75 f7           JNZ        MEM_CPY
```

`MOV EDX, DWORD PTR [EBX + 0X54]` - Get the SizeOfHeaders value from the PE header.

Once the size of headers is determined, there is a loop that copies that number of bytes from the beginning of the original PE file into the allocated memory for the new image.

Step 6: Copy each section to the new image

The PE header contains a field denoting the number of sections as well as the size of the optional header. The first section entry is located immediately following the optional headers. The section header has information on each section such as the virtual address, the raw address and the size. The virtual address is where the section will be when loaded into memory for execution, and the raw address is where the section is actually located on disk. These values are different for a reason. The sections have different permissions, for example, the .text section needs to be executable, while the rdata section needs to be read only. In order to give sections different permissions, they must be broken by page boundaries. But, in order to save space, when stored on disk, they do not need to be separated by page boundaries.

```
                         LAB_004175c6
004175c6 0f b7 53 06     MOVZX     EDX,word ptr [EBX + 0x6]
004175ca 0f b7 43 14     MOVZX     EAX,word ptr [EBX + 0x14]
004175ce 85 d2           TEST      EDX,EDX
004175d0 74 35           JZ        LAB_00417607
004175d2 8d 48 2c        LEA       ECX,[EAX + 0x2c]
004175d5 8b 45 fc        MOV       EAX,dword ptr [EBP + local_8]
004175d8 03 cb           ADD       ECX,EBX

                         LAB_004175da
004175da 8b 31           MOV       ESI,dword ptr [ECX]
004175dc 4a              DEC       EDX
004175dd 8b 59 fc        MOV       EBX,dword ptr [ECX + -0x4]
004175e0 03 f0           ADD       ESI,EAX
004175e2 89 55 d8        MOV       dword ptr [EBP + TEMP_VAR],EDX
004175e5 8b 51 f8        MOV       EDX,dword ptr [ECX + -0x8]
004175e8 03 d7           ADD       EDX,EDI
004175ea 85 db           TEST      EBX,EBX
004175ec 74 0c           JZ        LAB_004175fa
```

The first step is to find the section table.

`MOVZX EDX, WORD PTR [EBX + 0X6]` - Get the NumberOfSections from the PE header.
`MOVZX EAX, WORD PTR [EBX + 0X14]` - Get the SizeOfOptionalHeader from the PE header.

`LEA ECX, [EAX + 0X2C]`
`ADD ECX, EBX`
`MOV ESI, DWORD PTR [ECX]` - Get the value of the raw address of the current section.

`MOV EBX, DWORD PTR [ECX - 0X4]` - Get the number of bytes of the current section.

`ADD EDX, ESI` - Get the address of the section from the in memory module.

`MOV EDX, DWORD PTR [ECX - 0X8]`
`ADD EDX, EDI` - Get the virtual address of the section and add that to the memory base of the loaded image.

```
                    MEM_CPY
004175ee 8a 06          MOV        AL,byte ptr [ESI]
004175f0 88 02          MOV        byte ptr [EDX],AL
004175f2 42             INC        EDX
004175f3 46             INC        ESI
004175f4 4b             DEC        EBX
004175f5 75 f7          JNZ        MEM_CPY
```

Copy that section one byte at a time from the raw address location to the virtual address location.

```
                    LAB_004175fa
004175fa 8b 55 d8       MOV        EDX,dword ptr [EBP + TEMP_VAR]
004175fd 83 c1 28       ADD        ECX,0x28
00417600 85 d2          TEST       EDX,EDX
00417602 75 d6          JNZ        NEXT_SECTION
```

`ADD ECX, 0X28` - Go to the next raw address in the section table.

This will loop through the section table until the number of sections remaining has been set to 0.

Step 7: Populate the import table

```
00417607 8b b3 80       MOV        ESI,dword ptr [EBX + 0x80]
         00 00 00
0041760d 03 f7          ADD        ESI,EDI
0041760f 89 75 e8       MOV        dword ptr [EBP + ImportTable],ESI
```

`MOV ESI, DWORD PTR [EBX + 0X80]`
`ADD ESI, EDI` - EBX contains the offset of the PE header, so this is getting the RVA of the import table and converting it to the actual in memory address.

```
00417617 8b 46 0c       MOV        EAX,dword ptr [ESI + 0xc]
0041761a 03 c7          ADD        EAX,EDI
0041761c 50             PUSH       EAX
0041761d ff 55 e4       CALL       dword ptr [EBP + LoadLibrary]
```

`MOV EAX, DWORD PTR [ESI + 0XC]`
`ADD EAX, EDI` - Find the RVA of the pointer to the ascii name of the dll being loaded in this entry into the import table and get the in memory location.

Call LoadLibraryA to load this dll into memory.

```
00417627 8b 5e 10       MOV      EBX,dword ptr [ESI + 0x10]
0041762a 8b 06          MOV      EAX,dword ptr [ESI]
0041762c 03 df          ADD      EBX,EDI
0041762e 03 c7          ADD      EAX,EDI
```

The above will get the in memory address of the Import Address Table (IAT), `ESI + 0X10`, as well as the Import Lookup Table, `ESI`.

Now that we have the library loaded and the tables that we need, we will loop through the Import Lookup Table:

```
0041763b 85 c0          TEST     EAX,EAX
0041763d 74 22          JZ       GET_BY_NAME
0041763f 8b 08          MOV      ECX,dword ptr [EAX]
00417641 85 c9          TEST     ECX,ECX
00417643 79 1c          JNS      GET_BY_NAME
00417645 8b 46 3c       MOV      EAX,dword ptr [ESI + 0x3c]
00417648 0f b7 c9       MOVZX    ECX,CX
0041764b 8b 44 30 78    MOV      EAX,dword ptr [EAX + ESI*0x1 + 0x78]
0041764f 2b 4c 30 10    SUB      ECX,dword ptr [EAX + ESI*0x1 + 0x10]
00417653 8b 44 30 1c    MOV      EAX,dword ptr [EAX + ESI*0x1 + 0x1c]
00417657 8d 04 88       LEA      EAX,[EAX + ECX*0x4]
0041765a 8b 04 30       MOV      EAX,dword ptr [EAX + ESI*0x1]
0041765d 03 c6          ADD      EAX,ESI
0041765f eb 0c          JMP      LAB_0041766d

                    GET_BY_NAME                              XREF[
00417661 8b 03          MOV      EAX,dword ptr [EBX]
00417663 83 c0 02       ADD      EAX,0x2
00417666 03 c7          ADD      EAX,EDI
00417668 50             PUSH     EAX
00417669 56             PUSH     ESI
0041766a ff 55 e0       CALL     dword ptr [EBP + GetProcAddress]
```

```
MOV ECX, DWORD PTR [EAX]
TEST ECX, ECX
JNS GET_BY_NAME:
```

Get the next value in the Import Lookup Table. This value will either be an ordinal or the address of an ascii string. If the number is positive, it will jump to the part of the code that deals with an ascii string, otherwise it will deal with it as an ordinal.

Ordinal:

The code at addresses 0x417645 - 0x41765d will do the following:

- Get the PE header location of the imported library, located at BASE + 0x3C
- Get the word value of the ordinal.
- Get the RVA of the export table of the library. This is located at PE header + 0x78,
- Get the value of the base for the export table, this is the number of the first ordinal. It is located at Export Table + 0x10. Subtract that value from the ordinal searching for to get the offset into the address table.
- Get the value of the Export Address Table RVA. This is located at Export Table + 0x1C.
- Get the value at the offset of this ordinal in the Export Address Table. This value is an RVA; add it to the base of the module.

By name:

This will just get the ascii value of the function and call GetProcAddress.

```
0041766d 89 03          MOV     dword ptr [EBX],EAX
0041766f 83 c3 04       ADD     EBX,0x4
00417672 8b 45 ec       MOV     EAX,dword ptr [EBP + VirtualAlloc]
00417675 85 c0          TEST    EAX,EAX
00417677 74 06          JZ      LAB_0041767f
00417679 83 c0 04       ADD     EAX,0x4
0041767c 89 45 ec       MOV     dword ptr [EBP + VirtualAlloc],EAX

                   LAB_0041767f                                XRE
0041767f 83 3b 00       CMP     dword ptr [EBX],0x0
00417682 75 b7          JNZ     NEXT_PROC
```

After the address has been found, it will overwrite the current value in the IAT. It will then increment to the next value in both tables and continue the process until all procedures from this module have their addresses stored in the IAT.

```
00417687 83 c6 14        ADD        ESI,0x14
0041768a 89 75 e8        MOV        dword ptr [EBP + ImportTable],ESI
0041768d 83 3e 00        CMP        dword ptr [ESI],0x0
00417690 75 85           JNZ        NEXT_MODULE
```

After that module is complete, go to the next element in the Import Table by adding 0x14 to the current value. Loop through these until the next entry is NULL.

Step 8: Resolve the relocation table

The relocation table stores the method of converting hard coded data locations into the in memory address they will have when the module is loaded.

```
00417692 8b 5d f0        MOV        EBX,dword ptr [EBP + PE_HEADER]

                     LAB_00417695
00417695 8b c7           MOV        EAX,EDI
00417697 2b 43 34        SUB        EAX,dword ptr [EBX + 0x34]
0041769a 83 bb a4        CMP        dword ptr [EBX + 0xa4],0x0
         00 00 00 00
004176a1 89 45 d8        MOV        dword ptr [EBP + TEMP_VAR],EAX
004176a4 0f 84 aa        JZ         RELOCATIONS_COMPLETE
```

`MOV EAX, EDI`
`SUB EAX, DWORD PTR [EBX + 0X34]` - Get the value of the loaded image and store that into EAX. Get the value of the ImageBase, which is at the PE header + 0x34. All hard coded values are relative to the image base. Subtract the loaded image base from the hard coded ImageBase for relocation.

`CMP DWORD PTR [EBX + 0XA4], 0X0` - The value at EBX + 0xA4 is the number of entries in the relocation table. If there are no entries, nothing needs to be done.

```
004176aa 8b b3 a0       MOV        ESI,dword ptr [EBX + 0xa0]
         00 00 00
004176b0 03 f7          ADD        ESI,EDI
004176b2 89 75 e0       MOV        dword ptr [EBP + RELOCATION_DIR],ESI
004176b5 8d 4e 04       LEA        ECX,[ESI + 0x4]
004176b8 8b 01          MOV        EAX,dword ptr [ECX]
004176ba 89 4d e4       MOV        dword ptr [EBP + NUM_RELOCS_IN_SECTION],ECX
004176bd 85 c0          TEST       EAX,EAX
004176bf 0f 84 8f       JZ         RELOCATIONS_COMPLETE
```

`MOV ESI, DWORD PTR [EBX + 0XA0]`
`ADD ESI, EDI` – Get the in memory address of the relocation table.

`LEA ECS, [ESI + 0X4]`
`MOV EAX, DWORD PTR [ECX]` - Find out the block size of this relocation section.

```
004176c8 8b 16          MOV        EDX,dword ptr [ESI]
004176ca 83 c0 f8       ADD        EAX,-0x8
004176cd 03 d7          ADD        EDX,EDI
004176cf d1 e8          SHR        EAX,1
004176d1 89 45 d8       MOV        dword ptr [EBP + TEMP_VAR],EAX
004176d4 8d 46 08       LEA        EAX,[ESI + 0x8]
004176d7 89 45 e8       MOV        dword ptr [EBP + ImportTable],EAX
004176da 74 60          JZ         LAB_0041773c
004176dc 8b 7d d8       MOV        EDI,dword ptr [EBP + TEMP_VAR]
004176df 8b f0          MOV        ESI,EAX
```

Get the number of entries by taking the block size, subtracting 8 and dividing by 2. Subtracting 8 is necessary because the RVA and Block Size are included in the value. The number is divided by 2 because each entry contains 2 bytes.

`ADD EDX, EDI`
`LEA EAX, [ESI + 0X8]` – Get the in memory address of the first entry in the relocation table.

```
                    NEXT_RELOCATION
004176e1 0f b7 0e        MOVZX      ECX,word ptr [ESI]
004176e4 4f              DEC        EDI
004176e5 66 8b c1        MOV        AX,CX
004176e8 66 c1 e8 0c     SHR        AX,0xc
004176ec 66 83 f8 0a     CMP        AX,0xa
004176f0 74 06           JZ         LAB_004176f8
004176f2 66 83 f8 03     CMP        AX,0x3
004176f6 75 0b           JNZ        LAB_00417703
```

Get the next 2 byte value in the relocation table and shift right by 12 to get the first 4 bits. These bits represent the base relocation type. If this value is 3 that means this is a 32-bit relocation. If the value is 0xA, that means it is a 64-bit relocation. If the value is anything else other than 1 or 2, nothing will be done.

```
                    LAB_004176f8
004176f8 81 e1 ff        AND        ECX,0xfff
         0f 00 00
004176fe 01 1c 11        ADD        dword ptr [ECX + EDX*0x1],EBX
00417701 eb 27           JMP        LAB_0041772a
```

If it is a 32 or 64-bit relocation, the first thing it will do is take the 2 byte value and get the 12 least significant bits. These denote the offset within the relocation region of the value that needs to be changed. The amount changed by is the offset previously calculated based on the difference between the loaded address and the ImageBase. This will add that value to the value at the given offset. It will then loop through the rest of the regions doing the same.

```
                    LAB_00417703
00417703 66 3b 45 d4     CMP         AX,word ptr [EBP + local_30]
00417707 75 11           JNZ         LAB_0041771a
00417709 81 e1 ff        AND         ECX,0xfff
         0f 00 00
0041770f 8b c3           MOV         EAX,EBX
00417711 c1 e8 10        SHR         EAX,0x10
00417714 66 01 04 11     ADD         word ptr [ECX + EDX*0x1],AX
00417718 eb 10           JMP         LAB_0041772a

                    LAB_0041771a
0041771a 66 3b 45 d0     CMP         AX,word ptr [EBP + local_34]
0041771e 75 0a           JNZ         LAB_0041772a
00417720 81 e1 ff        AND         ECX,0xfff
         0f 00 00
00417726 66 01 1c 11     ADD         word ptr [ECX + EDX*0x1],BX
```

In the first case above, the relocation type is 1. In this case, instead of using the entire offset for relocation, it only uses the high 16 bits of value. In the second case, the relocation type is 2. In this case, it only uses the low 16 bits of the offset for relocation.

```
0041772a 6a 02           PUSH        0x2
0041772c 58              POP         EAX
0041772d 03 f0           ADD         ESI,EAX
0041772f 85 ff           TEST        EDI,EDI
00417731 75 ae           JNZ         NEXT_RELOCATION
```

It will then increment the value in the table by 2 and loop to the next value in that section, repeating the above process.

```
00417733 8b 7d f4       MOV      EDI,dword ptr [EBP + NEW_IMAGE]
00417736 8b 75 e0       MOV      ESI,dword ptr [EBP + RELOCATION_DIR]
00417739 8b 4d e4       MOV      ECX,dword ptr [EBP + BLOCK_SIZE]

                     LAB_0041773c                              XREF[1]:
0041773c 03 31          ADD      ESI,dword ptr [ECX]
0041773e 89 75 e0       MOV      dword ptr [EBP + RELOCATION_DIR],ESI
00417741 8d 4e 04       LEA      ECX,[ESI + 0x4]
00417744 8b 01          MOV      EAX,dword ptr [ECX]
00417746 89 4d e4       MOV      dword ptr [EBP + BLOCK_SIZE],ECX
00417749 85 c0          TEST     EAX,EAX
0041774b 0f 85 77       JNZ      NEXT_RELOCATION_TABLE_ENTRY
```

Once a section is complete, it will get the location of the section that was just processed and add to that the block size of that section. It will then loop through all of the sections until it comes to one whose size is 0.

Step 9: Call the entry point

```
                     RELOCATIONS_COMPLETE
00417754 8b 73 28       MOV      ESI,dword ptr [EBX + 0x28]
00417757 6a 00          PUSH     0x0
00417759 6a 00          PUSH     0x0
0041775b 6a ff          PUSH     -0x1
0041775d 03 f7          ADD      ESI,EDI
0041775f ff 55 dc       CALL     dword ptr [EBP + local_28]
00417762 33 c0          XOR      EAX,EAX
00417764 6a 00          PUSH     0x0
00417766 40             INC      EAX
00417767 50             PUSH     EAX
00417768 57             PUSH     EDI
00417769 ff d6          CALL     ESI
```

Find the entry point which is located at the PE header + 0x28. Call NtFlushInstructionCache to ensure that the memory that was overwritten is reloaded before execution. Call the entry point.

References:

(1) PE Header: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#optional-header-data-directories-image-only

(2) PEB: https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb

(3) PEB_LDR_DATA: https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb_ldr_data